1 Component-Interaction Automata Approach (CoIn)

Barbora Zimmerova*, Pavlína Vařeková**, Nikola Beneš**,
Ivana Černá***, Luboš Brim†, Jiří Sochor**

Masaryk University, Brno, Czech Republic

1.1 Introduction

The aim of the *CoIn approach (Component-Interaction Automata approach)* is to create a framework for formal analysis of behavioural aspects of large scale component-based systems. For the modelling purpose, we use the *Component-interaction automata* language [1]. For the verification, we employ a parallel model-checker DiVinE [2], which is able to handle very large, hence more realistic, models of component-based systems. Verified properties, like consequences of service calls or fairness of communication, are expressed in an extended version of the Linear Temporal Logic CI-LTL.

1.1.1 Goals and scope of the component model

The *Component-interaction automata* model behaviour of each component (both basic and composite) as a labelled transition system. The language builds on a simple, yet very powerful, composition operator that is able to reflect hierarchical assembly of the system and various communication strategies among components. Thanks to this operator, the language can be used for modelling the behaviour of components designed for or implemented in various component frameworks and models.

1.1.2 Modeled cutout of CoCoME

While modelling the CoCoME, we have focused on the aspect of communicational behaviour of components. Hence we did not treat aspects like non-functional properties or data manipulation. However in terms of component interaction, we have modelled the CoCoME completely in fine detail, based on the provided Java implementation of the system. We modelled also parts like GUI, Event Channels, or the Product Dispatcher component. The final model was verified using the DiVinE verification tool. We have checked the compliance of the model to the Use Cases from chapter 3, we have verified the Test Cases, and various other CI-LTL properties.

* Supported by the Czech Science Foundation within the project No. 102/05/H050.
** The authors have been supported by the grant No. 1ET400300504.
*** The author has been supported by the grant No. GACR 201/06/1338.
† The author has been supported by the grant No. 1ET408050503.

1.1.3 Benefit of the modeling

One of our main benefits is that we use a general formal modelling language that is, thanks to its flexible composition operator, able to capture behaviour of various kinds of component-based systems. The created model is in fact a labelled transition system, which is a format that can be directly verified using a large variety of existing methods.

1.1.4 Effort and lessons learned

For the full modelling of the CoCoME, we needed two person months. The verification was then performed automatically using the DiVinE tool. This exercise helped us to discover the limits and capabilities of our modelling language and verification methods. In particular, we have found general solutions to various modelling issues, like creation and destruction of instances, modelling of global shared state, or exception handling. In terms of verification, we have examined the efficiency of our verification methods on a large model of a real system.

1.2 Component Model

Our framework is represented by the *Component-interaction automata* language [1, 3]. It should be emphasized that Component-interaction automata are not meant to support implementation of component-based systems. They are intended as a modelling language that can be used to create detailed models of behaviour of component-based systems to support their formal analysis.

The language is very general, which follows from two things. First, the Component-interaction automata language does not explicitly associate *action names* used in the labels of automata with interfaces/services/events/etc., which allows the designers to make the association themselves. The association must only respect that if the same action name is used in two components, in one as an input and in the other one as an output, it marks a point on which the components may communicate. Second, the language defines one flexible *composition operator* that can be parametrized to simulate several communication strategies used in various component models. In this manner, Component-interaction automata can be instantiated to a particular component model by fixing the *composition operator* and *semantics of actions*.

1.2.1 Definition of a component-interaction automaton

Component-interaction automata capture each component as a labelled transition system with structured labels (to remember components which communicated on an action) and a hierarchy of component names (which represents the architectural structure of the component).

A *hierarchy of component names* is a tuple $H = (H_1, \ldots, H_n)$, $n \in \mathbb{N}$, of one of the following forms, S_H denotes the set of component names corresponding to

H. The first case is that $H_1, \ldots, H_n$ are pairwise different natural numbers; then $S_H = \bigcup_{i=1}^{n} \{H_i\}$. The second case is that $H_1, \ldots, H_n$ are hierarchies of component names where $S_{H_1}, \ldots, S_{H_n}$ are pairwise disjoint; then $S_H = \bigcup_{i=1}^{n} S_{H_i}$.

A *component-interaction automaton* (or a *CI automaton* for short) is a 5-tuple $\mathcal{C} = (Q, Act, \delta, I, H)$ where Q is a finite set of states, Act is a finite set of *actions*, $\Sigma = ((S_H \cup \{-\}) \times Act \times (S_H \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$ is a set of *labels*, $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*, $I \subseteq Q$ is a nonempty set of *initial states*, and H is a hierarchy of component names.

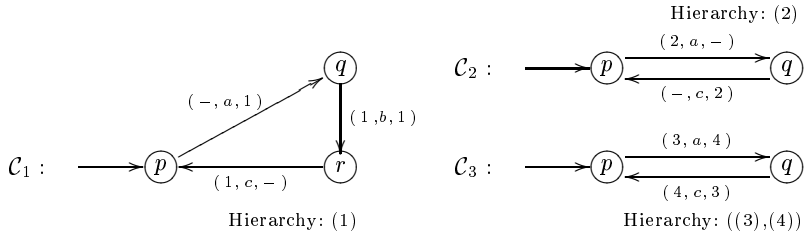


Fig. 1. Examples of CI automata

The labels have semantics of input, output, or internal, based on their structure. In the triple, the middle item represents an action name, the first item represents a name of the component that outputs the action, and the third item represents a name of the component that inputs the action. Examples of three CI automata are in figure 1. Here $(-, a, 1)$ in $\mathcal{C}_1$ signifies that a component with a numerical name 1 inputs an action a, $(1, c, -)$ in $\mathcal{C}_1$ signifies that a component 1 outputs an action c, $(4, c, 3)$ in $\mathcal{C}_3$ represents an internal communication of components 4 (sender) and 3 (receiver) on c, and $(1, b, 1)$ in $\mathcal{C}_1$ an internal communication on b inside the component 1 (it abstracts from the names of the sub-components of 1 that participated in the communication).

1.2.2 Composition of component-interaction automata

CI automata (a set of them) can be composed to form a composite CI automaton. The language of Component-interaction automata allows us to compose any finite set of CI automata (not necessarily two of them as usual in related languages) that have disjoint sets of component names. This guarantees that each primitive component in the model of the system has a unique name, which we use to identify the component in the model.

Informal description of the composition follows. For formal definitions see [1]. You can consult the description with a composite automaton of $\{\mathcal{C}_1, \mathcal{C}_2\}$ in fig. 2; the automata $\mathcal{C}_1, \mathcal{C}_2$ are in fig. 1. The *states* of the composite automaton are items of the Cartesian product of states of given automata. Among the states, the syntactically possible transitions (we call the set of them a *complete transition space*) are all transitions capturing that either (1) one of the automata follows its original transition or (2) two automata synchronize on a complementary transition (input of the first one, output of the second one, both with the same

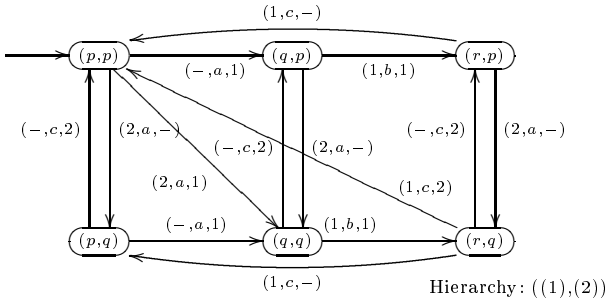


Fig. 2. A possible composition of the set $\{\mathcal{C}_1, \mathcal{C}_2\}$

action). Finally, we decide which of these transitions we really want to be part of the composite automaton, and remove the rest. The inscription $\mathcal{C} = \otimes_T\{\mathcal{C}_1, \mathcal{C}_2\}$, where T is a set of transitions, means that $\mathcal{C}$ is a composite automaton created by composition of $\mathcal{C}_1, \mathcal{C}_2$ and it consists of only those transitions from the complete transition space, that are included in T.

The automaton in fig. 2 is $\otimes_T\{\mathcal{C}_1, \mathcal{C}_2\}$ where T contains all transitions from the complete transition space. We write this as $\otimes\{\mathcal{C}_1, \mathcal{C}_2\}$. In addition to $\otimes$, along the text we use the following shortcuts for various types of $\otimes_T$. Denote Δ the complete transition space (set of all syntactically possible transitions) for the set of automata that enters the composition. Then

- $\otimes_{\bar{T}}$ is equivalent to $\otimes_{\Delta \setminus T}$
  The parameter T in this case specifies the *forbidden transitions*.
- $\otimes^{\mathcal{F}}$ is equivalent to $\otimes_T$ where $T = \{(q, x, q') \in \Delta \mid x \in \mathcal{F}\}$
  The parameter $\mathcal{F}$ specifies *allowed labels* – only the transitions with these labels may remain.
- $\otimes^{\bar{\mathcal{F}}}$ is equivalent to $\otimes_T$ where $T = \{(q, x, q') \in \Delta \mid x \notin \mathcal{F}\}$
  The parameter $\mathcal{F}$ specifies *forbidden labels* – no transition with any of these labels can remain in the composition.

Remember that the labels and transitions represent *items of behaviour* of the component-based system. Each internal label represents communication between two components, and each external label a demand for such communication. A transition signifies a specific communication (represented by the label) that takes place in the context given by the respective states.

1.2.3 Textual notation

For linear transcription of CI automata, we have defined the following textual notation, which was used for writing the CoCoME models. To introduce the notation, we rewrite some of the automata from the text above into the notation. The fig. 3 a) presents the automaton $\mathcal{C}_1$ from fig. 1. The keyword `automaton` states that we are going to define a new automaton by describing its state space. This is followed by the name of the automaton and the hierarchy of component names the automaton represents. The keyword `state` is followed by a list of states, `init` by a list of initial states, and `trans` by a list of transitions.

```
automaton C1 (1) {
  state p, q, r;
  init p;
  trans
    p -> q (-,a,1),
    q -> r (1,b,1),
    r -> p (1,c,-);
}
```

```
composite C {
  C1, C2;
}
```

```
composite C {
  C1, C2;
  restrictL
    (-,a,1), (2,a,-),
    (1,c,-), (-,c,2);
}
```

Fig. 3. a) Automaton $\mathcal{C}_1$ (left); **b)** Composition $\mathcal{C} = \otimes\{\mathcal{C}_1, \mathcal{C}_2\}$ (center); **c)** Composition $\mathcal{C} = \otimes^{\bar{\mathcal{F}}}\{\mathcal{C}_1, \mathcal{C}_2\}$ where $\mathcal{F} = \{(-, a, 1), (2, a, -), (1, c, -), (-, c, 2)\}$ (right)

The composite automaton $\mathcal{C} = \otimes\{\mathcal{C}_1, \mathcal{C}_2\}$ from fig. 2 is rewritten in fig. 3 b). No parameter T needs to be given here. For restricted composition we include a keyword representing the type of composition and the list of transitions or labels (see 1.2.2). Example of such composite is in fig. 3 c). For $\otimes_T$ the keyword is `onlyT` (only the **T**ransitions given), for $\otimes_{\bar{T}}$ the keyword is `restrictT`, for $\otimes^{\mathcal{F}}$ the keyword is `onlyL` (only the **L**abels given), for $\otimes^{\bar{\mathcal{F}}}$ the keyword is `restrictL`.

1.3 Modelling the CoCoME

In this section, we present our modelling approach on the CoCoME system, the *Trading System*, which we have modelled completely in fine detail (models available in [4]). First, we describe the input and output of the modelling process.

Input. The CoIn modelling process has two inputs: specification of the behaviour of primitive components and description of the static structure of the system (assembly and interconnection of components). The *specification of behaviour of primitive components* can be given e.g. as an informal description (used in preliminary design phase for estimating/predicting the future behaviour of the system) or as the implementation of the system (used for verification or formal analysis of an existing system). In our approach we use the implementation to derive CI automata of primitive components. The *static structure of the system* reveals the hierarchy of components, their interfaces and interconnections between them. This can be given in any syntactic form that allows a modeller to extract the information about the actions that can be communicated among the components and the restrictions on the communication, if any. The structural description is used to arrange the automata of primitive components into a hierarchy, and to parameterize the composition operator for each level of the hierarchy.

Output. The output of the modelling process is basically a CI automaton that represents the whole system (composition of the components that make up the system). In the CoIn approach, this automaton is used as an input of verification methods that will be discussed in section 1.5.

1.3.1 Static View

In our approach, the model of the static structure is an input (not an output) of the modelling process. In the case of this modelling example, we derive the

basic information about the components, their interconnection, and system's hierarchy from the UML component diagrams presented in chapter 3. The information about the actions that can be communicated among components is extracted from the Java implementation, namely from the interfaces of the classes. Fig. 4 reprints the hierarchical structure of the components and assigns each primitive component a unique numerical name that will be used to identify it. This diagram, in addition to the provided ones, includes the *ProductDispatcher* component that is also part of the implementation, and excludes the *Database* because we do not model it as a component. We decided to abstract from the Database, because the logic of data manipulation is already part of the *Data* component that we do model.

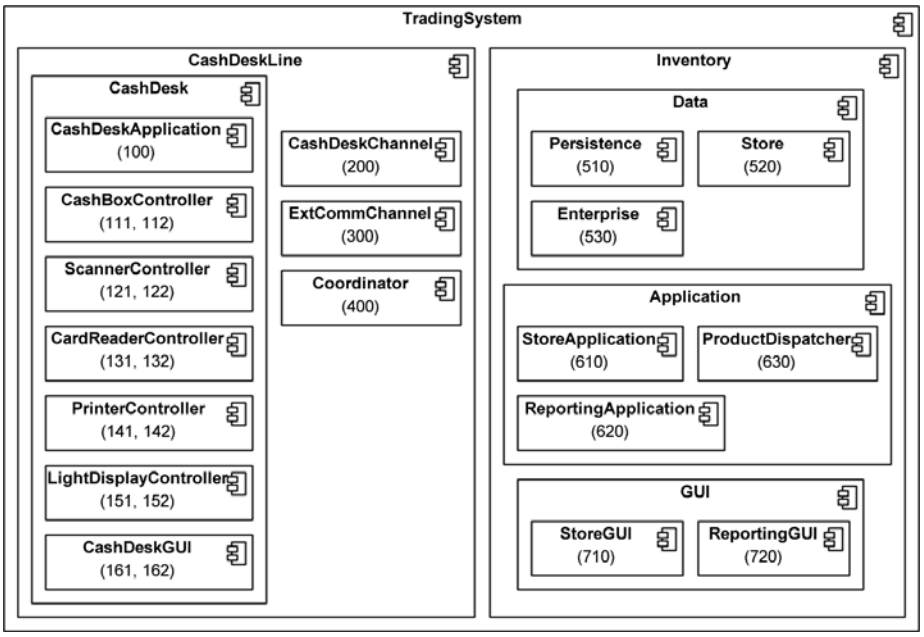


Fig. 4. Trading System overview

1.3.2 Behavioural View

For the behavioural view modelling, which is of our main interest, we use the *Component-interaction automata* language discussed in section 1.2. We decided to use the Java implementation as the main source for the modelling because it allows us to present a complex and interesting model of the system that is suitable for automatic analysis. Before we proceed to the models of the components, we present general modelling decision we did with respect to the implementation.

Initial decisions The components of the static view can be easily identified in the implementation as packages/classes. However we do not consider all pack-

ages and classes to be components. We omit modelling of classes that have no interesting interaction behaviour and act rather as data types than interaction entities. These include classes for different kinds of events, transfer objects and database entries. The same applies to Java library classes and supportive classes of the *Trading System*, like *DataIfFactory* or *ApplicationFactory*. Interfaces of the components and their parts can be identified as Java interfaces, resp. the sets of public methods constituting the parts.

We do not model the start of the system – the process of initial creation of all components and execution of their `main()` methods. We model the system from the point where everything is ready to be used. However we do model creation and destruction of the components that may be initialised during the run of the system, because in such case it is a part of system's behaviour.

Modelling components as CI automata The main idea for creating the automaton of a component is that we first capture component's behaviour on each service as a separate automaton, and then compose the automata for all the services into the model of the component. The services of components are represented by methods of the classes that implement the components. Therefore we will use the term *method* when referring to the implementation of a service that we model.

An automaton for a method. We assign each method, say `doIt()`, a tuple of actions where the first one represents the *call* of the method (we write it as `doIt`) and the second one the *return* (we write it as `doIt'`). These two determine the start and the end of the method's execution. Each method is modelled as a loop, which means that each of its behaviours starts and finishes in the same state, the initial one. Fig. 5 shows automata for two simple methods. The automaton $\mathcal{C}_1$ models a method `doA()` that does not call other methods to compute the result, the automaton $\mathcal{C}_2$ models a method `doB()` that calls the method `doA()` and when `doA()` returns, `doB()` also returns. More interesting examples follow in the text, see fig. 18 for an automaton of a complex method `changePrice()`.

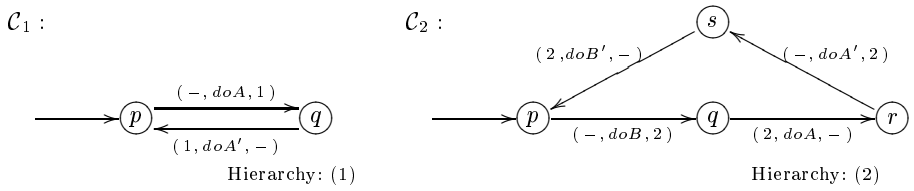


Fig. 5. Automata for simple methods

An automaton for a primitive (basic) component. Having automata for all the methods of a component, we can use the composition operator $\otimes_T$ to automatically create the automaton for the whole component. The composition operator allows us to compose the methods in various ways. When we apply $\otimes$, the resulting automaton simulates a component that can serve its methods (services) concurrently (modelled by interleaving of method parts). For example

see $\mathcal{C}_3 = \otimes\{\mathcal{C}_1, \mathcal{C}_2\}$ in fig. 6 (*doA*, *doB*, *computeA*, *computeB* are shortened to *dA*, *dB*, *cA*, *cB* respectively). As the component constitutes of two methods, its state space is in fact a two-dimensional grid. In case of three methods, it would be three-dimensional, and so forth. For this reason, we sometimes refer to this interleaving-like notion of composition as a *cube-like composition*.

Note, that the component $\mathcal{C}_3$ in fig. 6 may handle the methods `doA()` and `doB()` concurrently, but cannot accept a new call to a method that is currently executed. If we would like to model concurrent execution of several copies of one method, we would need to include the corresponding number of the automaton copies in the composition.

Another notion of composition that is common for composing models of methods, is the *star-like composition*, which reflects that from the initial state, the component may start executing any of its methods, but cannot start a new one before the previous method is finished. This can be found in GUI components where the calls on their (GUI-based) methods are serialized by the *event dispatching thread*. For example see $\mathcal{C}_4 = \otimes_{\bar{T}}\{\mathcal{C}_1, \mathcal{C}_2\}$ in fig. 6. The star-like notion of composition can be realized by the $\otimes_{\bar{T}}$ where T consists of the transitions that represent start of a method-call from other than the initial state.

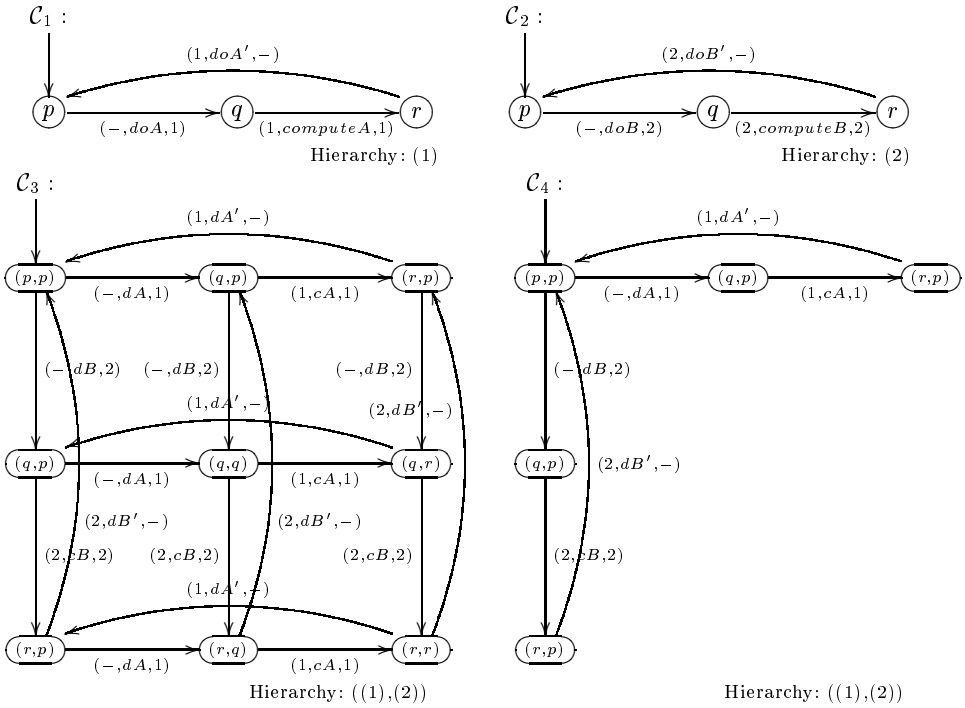


Fig. 6. Examples of the cube-like $\mathcal{C}_3$ and the star-like $\mathcal{C}_4$ composition of $\{\mathcal{C}_1, \mathcal{C}_2\}$

In a similar way, the composition gives us possibility to design other compositional strategies that would e.g. permit only specific ordering of services. For modelling of the CoCoME *Trading System* we use the cube-like and star-

like compositions most of the time to create the models of basic components by composing automata of their methods or other parts.

An automaton for a composite component. The situation is different for models of composite components. These are composed from the automata that usually interact with each other. Here, the most common notion of composition is the *handshake-like composition* which proceeds with a principle that anytime a component inside the composite is ready to perform an action that has its counterpart at one of the other components, it waits for this component to synchronize on the action. This kind of composition can be realized by $\otimes^{\mathcal{F}}$ where $\mathcal{F}$ consists of (1) all internal labels of components that enter the composition, (2) all new internal labels that may emerge from the composition, and (3) all external labels that have no counterpart at any other component. In the CoCoME modelling process, we use a modified version of this kind of composition where we manually, for each composite component, create $\mathcal{F}$ as a set consisting of (1) all internal labels of components that enter the composition, (2) only the new internal labels emerging from the composition *that are supported by a binding between the components* in the static view, and (3) all external labels *that represent methods provided/required on interfaces of the composite component* we are creating the model for, no matter if they synchronized inside the component or not.

Example. Imagine a *ComponentC* consisting of a *ComponentA* and *ComponentB* as in fig. 7. Suppose that the automaton of ComponentA has a set of labels $\mathcal{L}_A = \{ (-, serviceA, 1), (1, serviceA', -), (-, serviceC, 1), (1, serviceC', -), (1, serviceB, -), (-, serviceB', 1), (1, internalA, 1) \}$ and the ComponentB $\mathcal{L}_B = \{ (-, serviceB, 2), (2, serviceB', -), (2, serviceD, -), (-, serviceD', 2), (2, internalB, 2) \}$. Such sets of labels can be intuitively guessed from the figure. Then the automaton representing behaviour of ComponentC can be constructed as $AutomatonC = \otimes^{\mathcal{F}}\{AutomatonA, AutomatonB\}$ where $\mathcal{F} = \{ (1, internalA, 1), (2, internalB, 2), (1, serviceB, 2), (2, serviceB', 1), (-, serviceC, 1), (1, serviceC', -), (2, serviceD, -), (-, serviceD', 2) \}$.

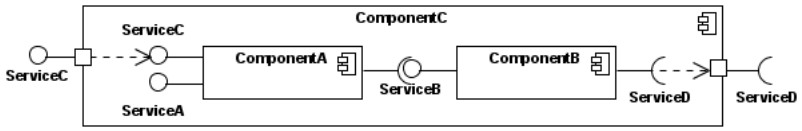


Fig. 7. A composite component

The static view given in fig. 7 is important here. Note that if the interfaces of ComponentA and ComponentB on `serviceB()` were not connected, the labels $(1, serviceB, 2), (2, serviceB', 1)$ would not represent feasible interactions and could not be part of $\mathcal{F}$. Analogically, the labels $(-, serviceA, 1)$, $(1, serviceA', -)$ cannot be part of $\mathcal{F}$ because `serviceA()` is not provided on the interface of ComponentC.

In the rest of this section, we demonstrate the capabilities of Component-interaction automata to model various kinds of interaction scenarios that we have faced when creating the model of the *Trading System*. We start with the models of primitive components (see fig. 4) and then present the composition anytime we have described all components needed for the composition. For space reasons, we do not describe all components completely. We just emphasize our solutions to interesting modelling issues when used for the first time. After reading this section, we believe that the reader would able to understand the whole CI automata model of the *Trading System*, even if some parts are not discussed here. The complete model of the *Trading System* is available in [4].

1.3.3 Specification of the CashDeskApplication (level 3)

The *CashDeskApplication* (component id *100*) is the main sub-component of the *CashDesk* composite component (see fig. 4). It is subscribed to receive events from two event channels, handles the events, and publishes new ones to control cash-desk devices. We start with the automata representing behaviour of its services, represented by the methods of the *ApplicationEventHandlerImpl* class. We have modelled all methods with only two exceptions. First, we omit the `onMessage()` method because it only delegates messages from event channels to `onEvent()` methods, which we decided to model as direct calls. Second, we skip the constructor `ApplicationEventHandlerImpl()` which is called only once when the system is started (and never during system's execution).

```
private void makeSale(PaymentMode mode) throws JMSException {
  SaleTO saleTO = new SaleTO();
  saleTO.setProductTOs(products);
  appPublisher.publish(topicSession
      .createObjectMessage(new SaleSuccessEvent()));
  externalPublisher.publish(topicSession
      .createObjectMessage(new AccountSaleEvent(saleTO)));
  externalPublisher.publish(topicSession
      .createObjectMessage(new SaleRegisteredEvent(
          topicName, saleTO.getProductTOs().size(), mode)));
}
```

Fig. 8. Java source of the `makeSale()` method

The component as a publisher The `makeSale()` (fig. 8, numerical name 2) is one of the methods that demonstrate the capability of the CashDeskApplication to publish events to event channels, the *CashDeskChannel (200)* and the *ExtCommChannel (300)* in particular. Each publication of an event (underlined in fig. 8) is identified by the event channel and the type of the event. We reflect both in the action names that we associate with each `publish()`. For instance the first `publish()` in fig. 8 is referred to as `publishSaleSuccessEvent` whereas the second one as `publishExtAccountSaleEvent`.

CI automata model of the method is in fig. 9. For short, *makeSale* is written as mS, *publishSaleSuccessEvent* as $pSSE$, *publishExtAccountSaleEvent* as $pEASE$, and *publishExtSaleRegisteredEvent* as $pESRE$.

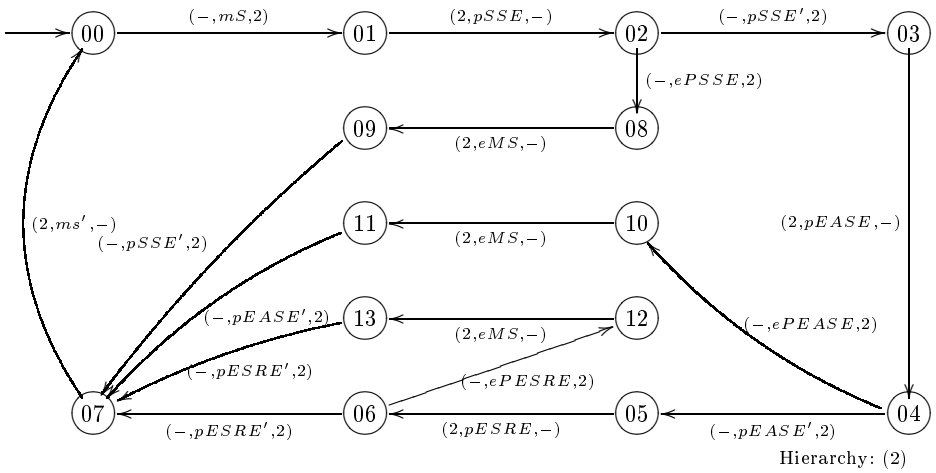


Fig. 9. CI automaton of the `makeSale()` method

First consider only the states $00 - 07$ and transitions among them. They represent three subsequent calls of `publish()` methods as one may expect from the Java code. The rest of the model captures that each of the `publish()` methods may throw an exception which is not caught by `makeSale()`. Consider the first publication. If an exception is thrown when `publishSaleSuccessEvent` is executed[1] (state 02), `makeSale()` synchronizes with it $(-, exceptionPublishSaleSuccessEvent, 2)$, written as $(-, ePSSE, 2)$, and moves to the state 08 where it forwards the exception $(2, exceptionMakeSale, -)$, finishes the publication and returns. We model `makeSale()` to forward the exception via catch and throw because it enables it to change its state and return when the exception occurs. Additionally, the exception is renamed because it makes it easier in the models of other methods to catch the exceptions of `makeSale()`.

The component as a subscriber The CashDeskApplication implements a variety of `onEvent()` methods, which handle the events the component is subscribed to. We assign each of them a unique tuple of action names reflecting the type of their argument to distinguish them. For instance the `onEvent(CashAmountEnteredEvent)` method (fig. 10) is referred `onEventCashAmountEntered`.

In fig. 10, the interesting behaviour is again underlined. It shows that the behaviour of the method differs significantly based on its argument. For this reason we decided to split this method into two, which can be done in our approach. We get `onEventCashAmountEntered`, which is called whenever a digit button is pressed, and `onEventCashAmountEntered_Enter`, which is called only when the Enter button is used. See fig. 11 for the automata of both of them. In the model, the *onEventCashAmountEntered* is shortened to *oECAE*, *onEventCashAmountEntered_Enter* to *oECAEE*, *publishChangeAmountCalculatedEvent* to *pCACE*, and *exceptionPublishChangeAmountCalculatedEvent* to *ePCACE*. If

[1] The automaton of the event channel performs $(200, exceptionPublishSaleSuccessEvent, -)$ when serving `publishSaleSuccessEvent`.

```
public void onEvent(CashAmountEnteredEvent cashAmountEnteredEvent) {
  ...
  if (currState.equals(PAYING_BY_CASH)) {
    switch (cashAmountEnteredEvent.getKeyStroke()) {
    case ONE: total = total.append("1"); break;
    case TWO: total = total.append("2"); break;
    ...
    case ENTER:
      try {
        appPublisher.publish(topicSession
            .createObjectMessage(new ChangeAmountCalculatedEvent(changeamount)));
        ...
        currState = PAID;
        ...
      } catch (JMSException e) {
        ...
      }
      break;
} } }
```

Fig. 10. Java source of the `onEvent(CashAmountEnteredEvent)` method

we did not want to split the method into two, we would finish with one model that non-deterministically allows for both behaviours.

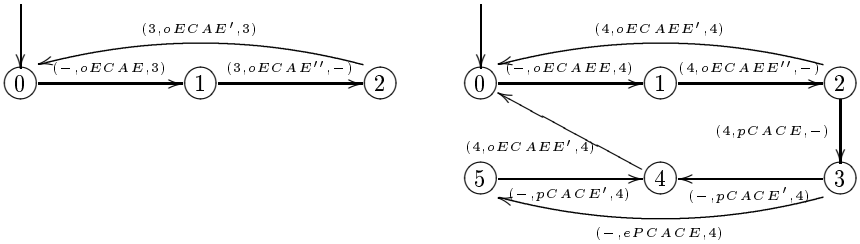


Fig. 11. CI automata of the `onEvent(CashAmountEnteredEvent)` method parts

The `onEvent()` methods are called asynchronously with a notification (represented by the action $onEventCashAmountEntered''$, or $oECAE''$ in the figure). Asynchronous methods start with getting the request as the synchronous ones $(-, onEventCashAmountEntered, 3)$, but returns internally because the caller is not interested in their response $(3, onEventCashAmountEntered', 3)$.

Exception handling The `onEvent(CashAmountEnteredEvent)` method (fig. 10) calls the `publish()` method inside a try block, which means that if the method throws an exception, `onEvent(CashAmountEnteredEvent)` catches it and moves to an appropriate catch block that handles the exception.

This principle is realized in the automaton of `onEventCashAmountEntered_Enter` (fig. 11 on the right). Here the execution after calling `publishChangeAmountCalculatedEvent` moves to the state 3 where it waits for `publish()` to return $(-, publishChangeAmountCalculatedEvent', 4)$ to get to the state 4. If an exception arrives before the return does $(-, exceptionPublishChangeAmountCalculatedEvent, 4)$, the execution moves to the state 5, confirms the return of

the publication, and possibly continues with the exception handling provided by the catch block, which is empty in this case. Then this exceptional branch joins the normal execution flow in state 4. The execution between this join and the end of the method is again empty here.

For more complex example of exception handling see the description of the `changePrice()` method in section 1.3.11.

Full model of a method provided on the interface Using the approach discussed above, we have created models of all methods constituting the CashDeskApplication. However, not all the methods are finally provided on the interface, some methods have only supportive character. One of the provided methods is the `onEvent(CashBoxClosedEvent)`, fig. 12, which uses two private methods – `makeSale()`, fig. 9, and `reset()`, which has the same structure as $\mathcal{C}_1$ in fig. 5 and a hierarchy (1).

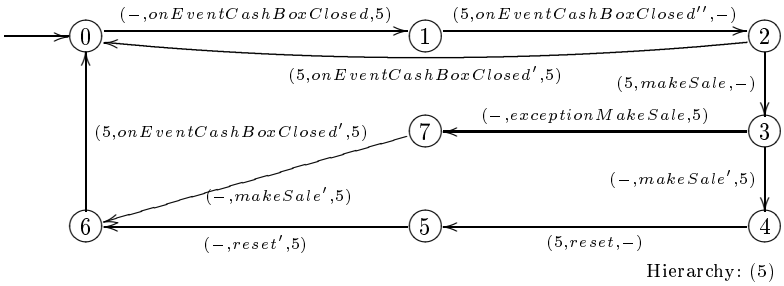


Fig. 12. CI automaton of the `onEvent(CashBoxClosedEvent)` method

The complete model of `onEvent(CashBoxClosedEvent)` is simply the composition of the automaton in fig. 12 with the automata of `makeSale()` and `reset()`. The composition is the *handshake-like* composition described in section 1.3.2. The composite automaton for `onEvent(CashBoxClosedEvent)` is in fig. 13. The actions are again shortened in the usual way.

Full model of the CashDeskApplication When we create the full models of all the methods available on the CashDeskApplication's interface (which are exactly all the `onEvent()` methods), we compose them into the automaton representing the model of the CashDeskApplication. In this case, we must use the *star-like* composition because the access to the `onEvent()` methods is controlled by a *Java session* that serializes them.

Note that the resulting model is a composite automaton where the basic units (represented by the component names in labels) are its methods. However, we are not interested in the interaction of methods, we want to model the interaction of primitive components, where the CashDeskApplication is one of them. Hence we transforms the automaton into a primitive one. This only means, that we rename all the component names used in the labels and transitions to a new numerical name 100 representing the CashDeskApplication, and change the hierarchy of component names to (100). See [1] for the definition of a primitive automaton

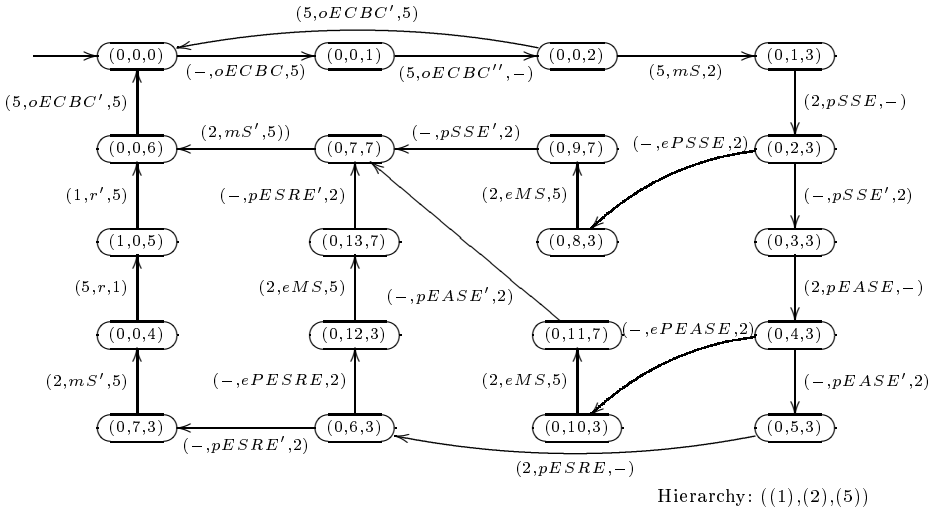


Fig. 13. Full model of the `onEvent(CashBoxClosedEvent)` method

and the relation of being primitive to a composite automaton. From now on, we suppose this final step of making the automaton primitive as implicit, and when presenting models of component parts, we use directly the numerical name of the resulting component in the labels.

Remember that by making an automaton primitive, we do not lose any information about its behaviours, we only lose the information about the names of component parts that participated in the behaviour.

Internal state of a component In this modelling example, we decided to determine the state of a component only with the interactions that the component is ready to perform. We do not take its internal state[2] into account. When a component decides, based on its state, whether it performs one interaction or another one, we model this basically as a non-deterministic choice. However this is just a modelling decision for the CoCoME. We are able to regard the internal state of a component (in case the number of internal states is finite). The basic idea of our approach follows.

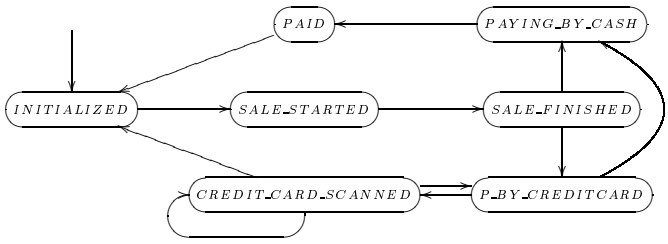


Fig. 14. Internal states of the CashDeskApplication

[2] We understand the internal state of a component as a current assignment of values to its attributes.

Suppose that the internal state of the CashDeskApplication is represented only by the states defined as *CashDeskStates* (see the implementation for details), which are: `INITIALIZED`, `SALE_STARTED`, `SALE_FINISHED`, `PAYING_BY_CREDITCARD`, `CREDIT_CARD_SCANNED`, `PAYING_BY_CASH`, `PAID`. Possible transitions between the states are depicted in fig. 14. Now consider for instance the `onEvent(CashAmountEnteredEvent)` method and its `onEventCashAmountEntered_Enter` model in particular (fig. 11 on the right). In its behaviour in between receive of the call and return, it either performs nothing, or tries to publish a *ChangeAmountCalculatedEvent*. The choice is non-deterministic here. However in reality, the choice is determined by the state of the component which can be seen in fig. 10. If we wanted to reflect the state, we would remodel `onEventCashAmountEntered_Enter` as in fig. 15.

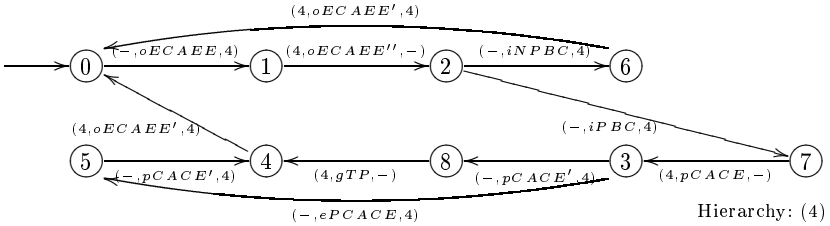


Fig. 15. Modified model of the `onEventCashAmountEntered_Enter`

The automaton first asks if the state equals to `PAYING_BY_CASH` – represented by $(-, isPAYING\_BY\_CASH, 4)$ and $(-, isNotPAYING\_BY\_CASH, 4)$ (in fig. 15 $(-, iPBC, 4)$ and $(-, iNPBC, 4)$) – and then follows the corresponding way. The automaton may also change the internal state of the component by asking it to do so via $(4, goToPAID, -)$, in the figure $(4, gTP, -)$. Last, the information about the component's state is added to the model of the CashDeskApplication itself. We do it by the *handshake-like* composition of the model of the CashDeskApplication with an additional automaton representing its states. The additional automaton is basically the automaton in fig. 14, where each transition is labeled with $(-, goToX, 1000)$, where X is the name of the state the transition is going to, and each state is additionally equipped with a self-transition with label $(1000, isX, -)$, where X is again the name of the state, and self-transitions with $(1000, isNotY, -)$ for all states Y different from X.

1.3.4 Specification of the CashBoxControllerComposite (level 3)

The *CashBoxController* component from fig. 4 is in fact a composite component that consists of two primitive components, the *CashBox (111)* and the *CashBoxController (112)*. Hence we refer to the composite as the *CashBoxControllerComposite* to avoid confusion. The *CashBox* component represents the user interface to the cash box device while the *CashBoxController* handles the communication of the CashBox with the rest of the system. We have assigned a different number to each of them, because they are distinguished also in the Use Cases presented in chapter 3.

CashBox (111). The CashBox consists of two kind of methods, the GUI-based methods that represent response to pressing a button by the Cashier, and regular methods that can be called by the rest of the system. These are `openCashBox()` and `closeCashBox()`. Methods of both kinds are modelled in a usual way. The difference between these two is that while `openCashBox()` and `closeCashBox()` can be called concurrently with other methods, the calls on the GUI-based methods are serialized by the event dispatching thread and hence no two GUI-based methods may execute at the same time.

In CI automata, we are able to respect this restriction when creating the composite automaton for the CashBox. We first create an auxiliary automaton CashBoxGUI with the star-like composition of GUI-based methods, and then compose this with the rest of the methods using the cube-like composition. This gives us exactly the result we want to have.

CashBoxController (112). The same applies to the *CashBoxController* model. Here, again two types of methods can be identified. The `onEvent()` methods, that are accessed serially, and the others that may be accessed in parallel. However in this case, there is only one `onEvent()` method, so we can apply the cube-like composition directly.

CashBoxControllerComposite. For the composition of the *CashBox* and *CashBoxController*, we use the handshake-like composition with synchronization on the `sendX()` methods. This composition results in the model of the *CashBoxControllerComposite* with only 555 states in total, even if the automaton of the *CashBoxController* has more than 390.000 states. This means that a composite component has nearly 1000-times less states than one of its sub-components. This is caused by the synchronization on common actions, which the handshake-like composition necessitates. In effect, the `sendX()` methods of the *CashBoxController* (modelled as callable in parallel) are serialized by the calls from the GUI-based methods of the *CashBox*, which are callable only in a serial order. Hence, all the states that represent concurrent execution of several `sendX()` methods disappear from the composition.

Models of the remaining CashDesk components, namely the *ScannerControllerComposite (121, 122)*, *CardReaderControllerComposite (131, 132)*, *PrinterControllerComposite (141, 142)*, *LightDisplayControllerComposite (151, 152)*, and *CashDeskGUIComposite (161, 162)* were created using the principles already discussed above. Therefore we decided to not to include their description here. The complete models of all of them can be found in [4].

1.3.5 Specification of the CashDesk (level 2)

The *CashDesk* component is a composite component that consists of all the components discussed above. Its model is a cube-like composition (with no restriction on transitions) because the components do not communicate with each other directly. They communicate via the *CashDeskChannel*, which is outside

the CashDesk component. As there is exactly one CashDeskChannel for each CashDesk, it would be better design decision to move the CashDeskChannel inside the CashDesk, which would make the CashDesk component more coherent while not influencing the model of the whole system. Only the hierarchy of component names would be different.

We decided to do it. We have created the model of the CashDesk using the handshake-like composition of the components discussed above with the CashDeskChannel (discussed below) with synchronization on the `publish()` and `onEvent()` methods that involve the CashDeskChannel, which can now happen only internally.

1.3.6 Specification of the CashDeskChannel (level 2)

The purpose of the *CashDeskChannel (200)* is in general to broadcast events that it receives from publishers to all subscribers that are interested in the events. The publication of an event is initiated by a publisher via calling `publish()`. The `publish()` method returns whenever the event is received by the channel. The channel then broadcasts the event to all subscribers that are interested in the event. We model this broadcast as a set of asynchronous calls to the `onEvent()` methods of the subscribers. These calls must be asynchronous to permit concurrent execution of the `onEvent()` methods. Based on this notes, the automaton for handling one event, namely the *SaleStartedEvent* (written as *SSEvent* for short) with three subscribers interested in it, is in fig. 16.

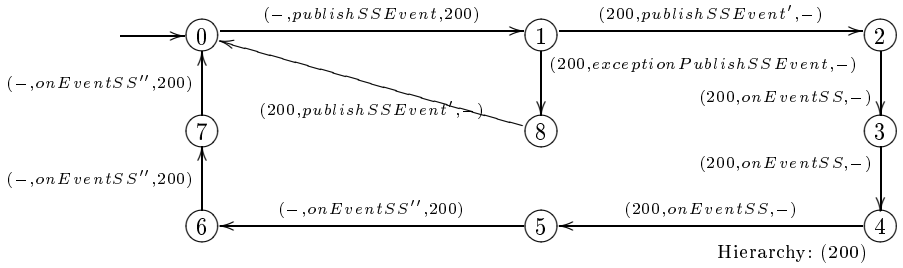


Fig. 16. CI automaton for `SaleStartedEvent`

The delivery of the event consists of two parts, the channel first initiates an `onEvent()` method of a subscriber $(200, onEventSaleStarted, -)$ and then waits for the notification from it $(-, onEventSaleStarted'', 200)$. The reason for this notification is only auxiliary. It guarantees that none of the subscribers acquires the event twice because it must first wait for the channel to take the notification from it. And the channel starts accepting the notifications after all the copies of the event are delivered. See fig. 11 for a model of an `onEvent()` method.

The drawback of this solution is that the channel needs to know the number of the subscribers in advance. If it was not possible to know the number in advance, we could model the transitions with $(200, onEventSaleStarted, -)$ and $(-, onEventSaleStarted'', 200)$ as two subsequent loops. However in such case we

could not guarantee that all the subscribers really get the event, because of not having the information of the actual number of them.

In the same way, we create an automaton for every event that can be handled by the channel, and compose them together using the cube-like composition.

The *ExtCommChannel (300)* is modelled in exactly the same way as the CashDeskChannel, therefore we do not discuss its model here.

1.3.7 Specification of the Coordinator (level 2)

The *Coordinator (400)* component is used for managing express checkouts. For this purpose, it keeps a list of sales that were done during last 60 minutes, which helps it to decide whether an express cash desk is needed. The component consists of two classes, the *CoordinatorEventHandler (410)* and the *Sale (420)*. Anytime a new sale arrives, the CoordinatorEventHandler creates a new instance of the Sale class to represent it in the list. Whenever the sale represented by an instance expires, the CoordinatorEventHandler removes the instance from the list which causes its destruction. We use this example to demonstrate, how we can model the dynamic creation and destruction of instances.

Before we explain our approach, remember that CI automata are finite state. This means, that we will never be able to model a system with unlimited number of instances. However this does not mean that we cannot verify such systems. In [5] we have proposed a verification approach addressing this issue. The solution there is based on getting the value k, dependent on the system and the property we want to verify, such that it guarantees that if we verify the system with $0, 1, 2, \ldots, k$ instances, we can conclude that the property holds on the system no matter how many instance are in use.

Now we come back to the modelling of the Coordinator component with a bounded number of Sale instances. For example let us set the maximum number of active instances to 10. First, we create the models of the Sale and the CoordinatorEventHandler in a usual way. In the model of the CoordinatorEventHandler we also model actions that cause the creation and destruction of Sale instances. In the case of creation, the code `new Sale(numberofitems, paymentmode, new Date())` is represented by the sequence $(410, Sale, -), (-, Sale', 410)$, in the case of destruction, the code `i.remove()` is represented by the sequence $(410, SaleD, -)$, $(-, SaleD', 410)$.

Second, we create a model of an *instance* of the Sale. It is an extension of the model of the Sale we have (fig. 17 on the left) with an initial activation part (fig. 17 on the right). The path $p \rightarrow q \rightarrow 0$ represents a call of the constructor of the component, and the way back $0 \rightarrow r \rightarrow p$ represents the destruction, which could start also from other states than 0. However it should not be possible for it to start in states p or q to guarantee that the system can destruct only the instances that have been created before.

The model of the Coordinator is a handshake-like composition of the set consisting of the model of the CoordinatorEventHandler and 10 copies of the automaton for the Sale instance. In the composite, all the Sale-instance automata

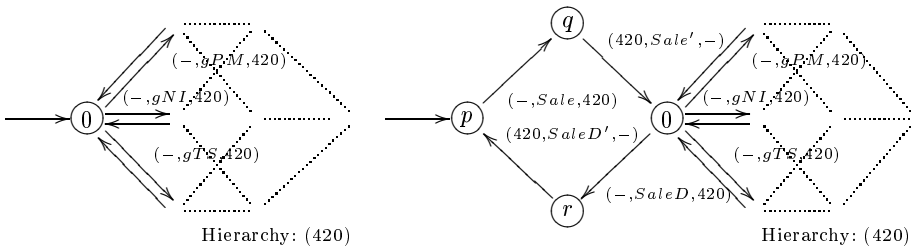


Fig. 17. CI automata of the *Sale class* and the *Sale instance*

start in the state p, which means that they are inactive at the beginning – the CoordinatorEventHandler cannot access their behaviour which starts to be available in state 0. However, when the CoordinatorEventHandler calls a constructor of the Sale $(410, Sale, -), (-, Sale', 410)$, one of the instances synchronizes with it and moves to the state that makes its functionality available. When the CoordinatorEventHandler removes the Sale from its list $(410, SaleD, -), (-, SaleD', 410)$, it returns back to the state p and waits for another call of its constructor. Note that if the Sale performed some calls inside its constructor or destructor, these calls could be included in the paths $p \to q \to 0$ and $0 \to r \to p$ as one may intuitively think.

We have experimented with different numbers of Sale instances in the system and realized that no matter how many of them we allow, the external behaviour of the Coordinator component remains the same. For this reason, we finally decided to regard the Sale class as something internal to the Coordinator and not to include it in the final model of the Coordinator.

1.3.8 Specification of the CashDeskLine (level 1)

The model of the *CashDeskLine* component is a composition of the CashDesk, ExtCommChannel and Coordinator components using the handshake-like composition.

1.3.9 Specification of the Persistence (level 3)

The *Persistence (510)* is the first component belonging to the *Inventory* part of the *Trading System*. It consists of three parts, that we interpret as subcomponents. That are *PersistenceImpl*, *PersistenceContextImpl* and *TransactionContextImpl*.

Both PersistenceContextImpl and TransactionContextImpl are modelled with the initial activation part, because they may be created and destructed. Their creation is managed by `getPersistenceContext()` (in the PersistenceImpl) and `getTransactionContext()` (in the PersistenceContextImpl) methods that call the constructors in their bodies. However their destruction is not managed inside the Persistence component. We will discuss the destruction when describing the *StoreApplication*, which uses these components.

The PersistenceContextImpl and TransactionContextImpl are used in the way that at any time, only one instance of each is needed to be active. Therefore

we create the model of the Persistence as a composition of the PersistenceImpl, one instance of the PersistenceContextImpl and one instance of the TransactionContextImpl. In section 1.5 you may see how we can verify that one instance of each is really sufficient.

The *Store (520)* and the *Enterprise (530)* components are created in a usual way. Hence we do not discuss them here.

1.3.10 Specification of the Data (level 2)

The *Data* component is a composition of the Persistence, Store and Enterprise components. However, there are not only one of each. The Data component is used by the *StoreApplication*, *ReportingApplication* and *ProductDispatcher* where each of them requires its own Persistence, Store and Enterprise according to the Java implementation. Hence also in the model, we include three copies of the Persistence (511, 512, 513), Store (521, 522, 523), and Enterprise (531, 532, 533). The copies differ only in the name of the automata and the component name that identifies them. Currently, the copies are created via *copy-paste*, however we aim to support the *type-instance* treatment in future.

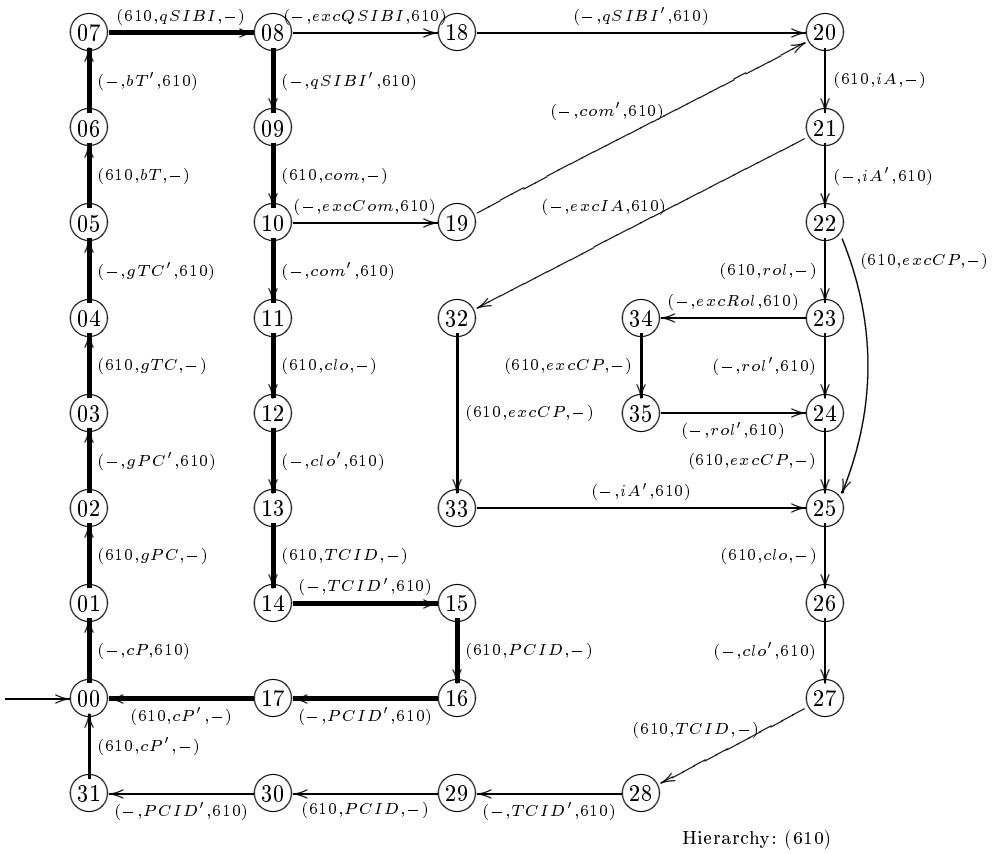
During the composition, we need to take care of the communication among these. For instance, the Store component calls the `getEntityManager()` method of the Persistence. And we want the first Store (521) to call just the first Persistence (511), event if it is syntactically possible to call also the second (512) and the third one (513). Our composition is able to respect such restrictions. We only state that the labels $(521, getEntityManager, 512)$, $(512, getEntityManager', 521)$, $(521, getEntityManager, 513)$, $(513, getEntityManager', 521)$ do not represent feasible behaviour of the composite system and hence all transitions with such labels must be removed during the composition.

1.3.11 Specification of the StoreApplication (level 3)

For the *StoreApplication (610)*, we present a model of one method to illustrate, what structure these methods have. It is the `changePrice()` method (automaton in fig. 18, implementation in fig. 19).

The automaton of the method (fig. 18) has 35 states. The states $00 - 17$ form the main execution flow (if no exception occurs), which is typeset in boldface. The execution starts with asking for a new instance of the PersistenceContextImpl by calling `getPersistenceContext()`, and analogically for a new instance of the TransactionContextImpl via `getTransactionContext()`. The `changePrice()`, which owns the only reference to these instances, loses the reference when its execution finishes. And so the instances get destructed eventually. We model this by calling `PersistenceContextImplD` and `TransactionContextImplD` at the end of each possible execution flow of the method.

The path between states $03 - 11$ corresponds to a try block in the code. If an exception occurs in any of these states (which may happen in 08 and 10), the execution of the methods that thrown the exception is finished and the flow goes



Action name	Abbreviation
changePrice	*cP*
exceptionChangePrice	*excCP*
getPersistenceContext	*gPC*
getTransactionContext	*gTC*
beginTransaction	*bT*
queryStockItemById	*qSIBI*
exceptionQueryStockItemById	*excQSIBI*
commit	*com*
exceptionCommit	*excCom*
close	*clo*
TransactionContextImplD	*TCID*
PersistenceContextImplD	*PCID*
isActive	*iA*
exceptionIsActive	*excIA*
rollback	*rol*
exceptionRollback	*excRol*

Fig. 18. CI automaton of the `changePrice()` method

```
public ProductWithStockItemTO changePrice(StockItemTO stockItemTO) {
  ProductWithStockItemTO result = new ProductWithStockItemTO();
  PersistenceContext pctx = persistmanager.getPersistenceContext();
  TransactionContext tx = null;
  try {
    tx = pctx.getTransactionContext();
    tx.beginTransaction();
    StockItem si = storequery.queryStockItemById(stockItemTO.getId(), pctx);
    si.setSalesPrice(stockItemTO.getSalesPrice());
    result = FillTransferObjects.fillProductWithStockItemTO(si);
    tx.commit();
  } catch (RuntimeException e) {
    if (tx != null && tx.isActive())
      tx.rollback();
    e.printStackTrace();
    throw e; // or display error message
  } finally {
    pctx.close();
  }
  return result;
}
```

Fig. 19. Java source of the `changePrice()` method

to the state 20 where the exception handling starts. The execution between states $20 - 25$ corresponds to the catch block in the code. Along this way two methods are called, `isActive()` and `rollback()`. Both of them may throw an exception. In such a case, the exception is not caught, it is propagated higher up. We model this as an output of a new exception that is caused by input of the existing one. If any of these paths finished correctly (without throwing an exception), the flow would continue to the finally block, which is represented by the states 11 – 13. However all paths resulting from the catch block do throw an exception, therefore all lead to an exceptional finally block (after which the method is finished immediately, whereas after the normal finally block the execution may continue). This is represented by the states $25 - 27$.

The model of the *ReportingApplication (620)* component is similar to the one of the StoreApplication, therefore we do not discuss it here.

1.3.12 Specification of the ProductDispatcher (level 3)

The *ProductDispatcher (630)* component manages the exchange of low running goods among stores. It uses an *AmplStarter* sub-component that computes optimizations, and itself implements a variety of methods to support the only method it provides on its interface, the `orderProductsAvailableAtOtherStores()`.

We comment only on the `markProductsAvailableInStockPD()` method, which calls a `markProductsAvailableAInStock()` method on selected Stores to mark the products that are a subject of product exchange. In the model, we are naturally not able to find out, what Stores should be addressed at the time the method is called. The Stores are selected by a complicated algorithm. For this reason, we model the call of Stores in a loop (see fig. 20), which may synchronize with any subset of the Stores that are currently active

in the model. This is a simple solution that does not assure that the Product-Dispatcher calls each Store at most once, but we decided to accept this over-approximation. In fig. 20 the *markProductsAvailableInStockPD* is shortened to *mPAISPD*, *computeProductMovements* to *cPM*, and *markProductsAvailableIn-Stock* to *mPAIS*.

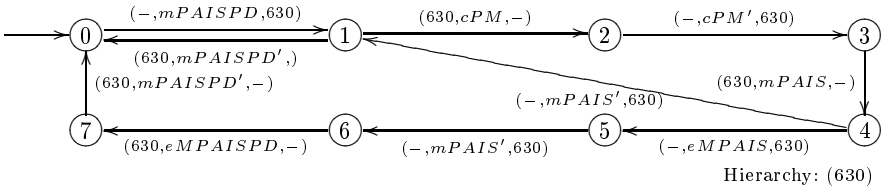


Fig. 20. CI automaton of the `markProductsAvailableAInStockPD()` method

The models of the *Application*, *GUI* and finally also *Inventory* composite components are created in a usual way.

1.3.13 Specification of the TradingSystem (level 0)

The last thing to do is to compose the CashDeskLine component and the Inventory component into the *TradingSystem* which is done using the handshake-like composition. The resulting model of the TradingSystem is not closed (containing internal transitions only). It has a lot of external transitions representing receive of input on the GUI of both CashDeskLine and Inventory. And it has a few external transitions representing communication with the Bank. We can decide if we want to analyse the TradingSystem as it is (no restriction on the usage profile), or create automata of the Cashier, Manager, and Bank and compose them with the TradingSystem into a closed model with internal transitions only. It is up to the designer to choose one of these strategies.

1.4 Transformations

In the context of the CoCoME, there are two types of transformations that can be helpful during the process. The first one is the *transformation of a composite automaton into a primitive one*. See the end of the section 1.3.3 for its description. This transformation is used as a final step of modelling of any primitive component that is created as a composition of its parts.

The second one is the *reduction of the model with respect to an equivalence that overlooks unimportant labels*. In [6] we have defined various kinds of such relations for CI automata. The relations are designed to indicate situations where a substitution of components preserves properties of the whole model. At the same time, they can be exploited to minimize the model by substituting a component by a smaller yet equivalent one. In the CoCoME project, we have not used this transformation because our tool was able to analyse the whole model. However

if the model was larger, we would use it to reduce the models of primitive components, which consist of a significant number of transitions representing inner interaction with private methods, which is not very important in the context of the whole system.

1.5 Analysis

Each modelling process is driven by the purpose the model is used for. In the case of our approach, the purpose is the automatic verification of temporal properties of component-based system. Verification can be used in the design phase to predict the behaviour of the system that we are going to assemble, or after it when we want to analyse properties of an existing system.

Moreover, verification techniques may help also in the modelling process to find modelling errors, or to evaluate whether the simplifications that were done during modelling do not introduce unexpected behaviours, like deadlocks. In the project, we found this very helpful when evaluating various modelling decisions.

In this section, we demonstrate the capability of the formal verification on analysis of the model created in the previous section. First, we focus on checking general temporal properties and on searching deadlock situations in the model. Next, we discuss how the use cases and test cases presented in chapter 3 can be evaluated on the model.

1.5.1 Temporal-logic properties

For verification of temporal properties, we use the model checking technique [7]. The input of the technique is the model of the system (a CI automaton in our case) and a temporal property we want to check. For properties specification, we use an extended version of the linear temporal logic LTL [8] which we refer to as *CI-LTL* along the text. CI-LTL is designed to be able to express properties about component interaction (i.e. labels in automata), but also about possible interaction (i.e. label enableness). Therefore, it is both state-based and action-based. CI-LTL uses all standard LTL operators, namely the next-step $\mathcal{X}$, and the until $\mathcal{U}$ operator, as well as the standard boolean operators. It has, however, no atomic propositions as their role is played by two operators, $\mathcal{E}(l)$ and $\mathcal{P}(l)$, where l is a label. Their meaning is informally given as follows.

- $\mathcal{E}(l)$ means "Label l is **E**nabled" and is true in all states of the system such that the interaction represented by label l can possibly happen.
- $\mathcal{P}(l)$ means "Label l is **P**roceeding" and is true whenever the interaction represented by label l is actually happening.

Syntax. Formally, for a given set of labels, formulas of CI-LTL are defined as

1. $\mathcal{P}(l)$ and $\mathcal{E}(l)$ are formulas, where l is a label.
2. If Φ and Ψ are formulas, then also $\Phi \wedge \Psi, \neg\, \Phi, \mathcal{X}\ \Phi$ and $\Phi\ \mathcal{U}\ \Psi$ are formulas.

3. Every formula can be obtained by a finite number of applications of steps (1) and (2).

Other operators can be defined as shortcuts: $\Phi \vee \Psi \equiv \neg\,(\neg\,\Phi \wedge \neg\,\Psi)$, $\Phi \Rightarrow \Psi \equiv \neg\,(\Phi \wedge \neg\,\Psi)$, $\mathcal{F}\,\Phi \equiv \text{true}\;\mathcal{U}\,\Phi$, $\mathcal{G}\,\Phi \equiv \neg\,\mathcal{F}\neg\,\Phi$, where $\mathcal{F}$ is the *future* and $\mathcal{G}$ the *globally* operator.

Semantics. Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be a CI automaton. We define a *run* of $\mathcal{C}$ as an infinite sequence $\sigma = q_0, l_0, q_1, l_1, q_2, \ldots$ where $q_i \in Q$, and $\forall i\,.\,(q_i, l_i, q_{i+1}) \in \delta$. We further define:

- $\sigma(i) = q_i$ (i-th state of σ)
- $\sigma^i = q_i, l_i, q_{i+1}, l_{i+1}, q_{i+2}, \ldots$ (i-th sub-run of σ)
- $\mathcal{L}(\sigma, i) = l_i$ (i-th label of σ)

CI formulas are interpreted over runs where the satisfaction relation $\models$ is defined inductively as

$$
\begin{aligned}
\sigma &\models \mathcal{E}(l) && \iff \exists q\,.\,\sigma(0) \xrightarrow{l} q \\
\sigma &\models \mathcal{P}(l) && \iff \mathcal{L}(\sigma, 0) = l \\
\sigma &\models \Phi \wedge \Psi && \iff \sigma \models \Phi \text{ and } \sigma \models \Psi \\
\sigma &\models \neg\,\Phi && \iff \sigma \not\models \Phi \\
\sigma &\models \mathcal{X}\,\Phi && \iff \sigma^1 \models \Phi \\
\sigma &\models \Phi\;\mathcal{U}\,\Psi && \iff \exists j \in \mathbb{N}_0\,.\,\sigma^j \models \Psi \text{ and } \forall k \in \mathbb{N}_0, k < j\,.\,\sigma^k \models \Phi
\end{aligned}
$$

Properties expressed in CI-LTL The logic enables us to specify many interesting properties about component-based systems. Let us consider the model resulting from the composition of the *Trading System* with a *Manager* who performs some operations on the Inventory. Here are some properties of the model we may want to check.

1. Whenever the *StoreApplication (610)* calls *getTransactionContext()* on the *Persistence (511)*, it gets a response at some point in the future. This can be used to check that a new instance of the *TransactionContextImpl* can be activated when demanded.

$$
\mathcal{G}\,\big(\mathcal{P}(610, \text{getTransactionContext}, 511) \Rightarrow \mathcal{F}\,\mathcal{P}(511, \text{getTransactionContext}', 610)\big)
$$

2. If the *Persistence (511)* receives a call from the *Store (521)*, it returns a result to it. And before it does so, it is not able to deliver the result to someone else - to *Store (522)*.

$$
\mathcal{G}\,\big(\mathcal{P}(521, \text{getEntityManager}, 511) \Rightarrow \mathcal{X}\,\big(\neg\,\mathcal{E}(511, \text{getEntityManager}', 522)\;\mathcal{U}\,\mathcal{P}(511, \text{getEntityManager}', 521)\big)\big)
$$

3. If the *StoreApplication (610)* starts a transaction with the *Persistence (511)*, it correctly closes the transaction before it is able to start another one.

$$\mathcal{G}\,\big(\mathcal{P}(610, \text{beginTransaction}, 511) \Rightarrow \mathcal{X}\ (\neg\,\mathcal{E}(610, \text{beginTransaction}, 511)\ \mathcal{U}\,\mathcal{P}(610, \text{close}, 511))\big)$$

Another type of interesting properties are deadlock situations. Deadlocks are the states from which in the model it is not possible to perform any step further. On the level of the system, they do not necessarily need to represent halting of the system. They may also reflect other kinds of failures, like breakdown, infinite cycling, or just return of a warning message. In the context of component-based systems, it is useful to check the model for *local* deadlocks, which we define as deadlocks of a single component. A deadlock of a component in the model may be very difficult to find, because even if a component cannot move, the system may continue its execution because other components are running. CI-LTL allows us to capture local deadlocks in the following way.

4. It cannot happen that the *StoreApplication (610)* is ready to call `queryStockItemById()` but never can do so because its counterpart is never ready to receive the call.

$$\mathcal{G}\,\big(\mathcal{E}(610, \text{queryStockItemById}, -) \Rightarrow \mathcal{F}\,\mathcal{E}(610, \text{queryStockItemById}, 521)\big)$$

5. It cannot happen that the *StoreApplication (610)* wants to begin a transaction, but the *Persistence (511)* is not *right in the current state* ready to serve it (stricter version of local deadlock).

$$\mathcal{G}\ \neg\ \big(\mathcal{E}(610, \text{beginTransaction}, -)\ \wedge\ \neg\ \mathcal{E}(610, \text{beginTransaction}, 511)\big)$$

For the above mentioned model all the properties were automatically verified with the help of the verification tool DiVinE (see section 1.6). The verification confirmed that the model has all listed properties.

1.5.2 Use cases

Chapter 3 presents a set of sequence diagrams showing the behavioural scenarios of the *Trading System* in terms of component interaction. However, we have created our model independently on these scenarios. Now we show how we can check that the model complies to them.

Each scenario is in fact a sequence of execution steps driven by the user-given inputs. If we describe the scenarios as sequences of CI-automata labels, the model-checking method enables us to automatically explore the model and find out whether the behaviours are present in it. We demonstrate the process on the **UC 1: ProcessSale :: CashPayment**. For all other use cases it can be done analogically.

First, we define the model that will be checked. It is a composition of the *Trading System* with the automaton representing the usage profile determined by

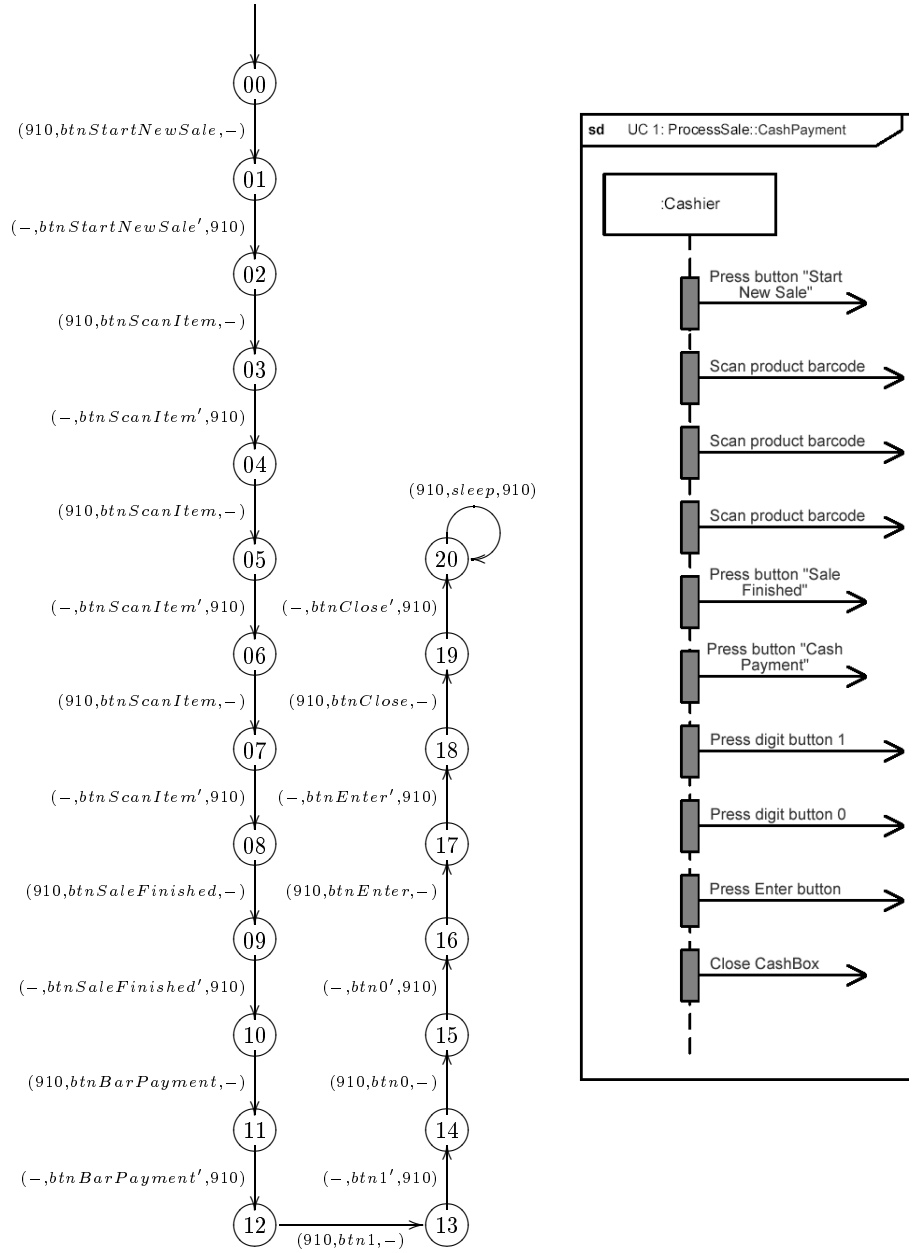


Fig. 21. Cashier

the Cashier (fig. 21). We model a finite behaviour of the cashier as an infinite one (with a loop in the end) not to introduce deadlock states into the composition.

Second, we capture the use case scenario of **UC 1: ProcessSale :: CashPayment** (chapter 3, fig. **??** and **??**) as a sequence of labels and check that such a behaviour is really present in the model. Part of the scenario is rewritten in table 1, where each line corresponds to a label in the sequence, e.g. the first line to the label $(910, btnStartNewSale, 111)$.

No.	Sender	Action	Receiver
1.	**Cashier (910)**	**btnStartNewSale**	CashBox (111)
2.	CashBox (111)	sendSaleStartedEvent	CashBoxController (112)
3.	CashBoxController (112)	publishSaleStartedEvent	CashDeskChannel (200)
4.	CashDeskChannel (200)	onEventSaleStarted	CashDeskApplication (100)
5.	CashDeskChannel (200)	onEventSaleStarted	PrinterController (142)
6.	CashDeskChannel (200)	onEventSaleStarted	CashDeskGUI (162)
7.	CashBoxController (112)	sendSaleStartedEvent'	CashBox (111)
8.	**Cashier (910)**	**btnScanItem**	Scanner (121)
9.	Scanner (121)	sendProductBarcodeScannedEvent	ScannerController (122)
10.	ScannerController (122)	publishProductBarcodeScannedEvent	CashDeskChannel (200)
	...	...	...
75.	**Cashier (910)**	**btnClose**	CashBox (111)
76.	CashBox (111)	sendCashBoxClosedEvent	CashBoxController (112)
77.	CashBoxController (112)	publishCashBoxClosedEvent	CashDeskChannel (200)
78.	CashDeskChannel (200)	onEventCashBoxClosed	CashDeskApplication (100)
79.	CashDeskChannel (200)	onEventCashBoxClosed	PrinterController (142)
80.	CashBoxController (112)	sendCashBoxClosedEvent'	CashBox (111)
81.	CashDeskApplication (100)	publishSaleSuccessEvent	CashDeskChannel (200)
82.	CashDeskChannel (200)	onEventSaleSuccess	PrinterController (142)
83.	CashDeskChannel (200)	onEventSaleSuccess	CashDeskGUI (162)
84.	CashDeskApplication (100)	publishExtAccountSaleEvent	ExtCommChannel (300)
85.	ExtCommChannel (300)	onEventExtAccountSale	StoreApplication (610)
86.	CashDeskApplication (100)	publishExtSaleRegisteredEvent	ExtCommChannel (300)
87.	ExtCommChannel (300)	onEventExtSaleRegistered	Coordinator (400)

Table 1. UC 1: ProcessSale :: CashPayment scenario

Last, we automatically verify that there is such a sequence present in the model, possibly interleaved with other labels (execution of other components or fine-grained interaction which is not captured on the sequence diagram). In case it is, the verification process reports a run which shows this. From the run, one can get the full interaction scenario which can be used later on to improve the sequence diagram of the use case.

1.5.3 Test cases

Besides the use cases, chapter 3 provides formalization of use case behaviour by way of test scenarios, which can be also evaluated on the model. The scenarios are of two types: the *informal* ones that prospect for the existence of a *good* behaviour, and the formal ones, given as Java test classes, that check that all behaviours of the system are *good* in some sense. For both of them, we formulate a corresponding CI-LTL formula that can be verified on the model.

Informal scenarios The informal scenarios are verified in a negative way. The formula states that all the behaviours are *bad*, and we want to prove it false – there is a behaviour that is not *bad*. We present this on the *ShowProductsForOrdering* scenario, which states that: *Store shall provide functionality to generate a report about products which are low on stock.*

This functionality is represented by the `getProductsWithLowStock()` method of the *StoreApplication (610)* component. So the formula is

$$\neg\, \mathcal{F}\, \mathcal{E}(610, \mathrm{getProductsWithLowStock}', -)$$

and it states that from the initial state of the model we cannot reach a transition that represents successful return of the `getProductsWithLowStock()` method. The verification reported a run that does not satisfy this and hence represents the *good* behaviour we have searched for.

Formal scenarios The Java test cases are specified by sequences of method calls that should be correctly processed by the system. This can be interpreted in two ways: the test passes if (1) the sequence of methods always finishes (possibly by throwing an exception), or (2) it always finishes *correctly* (with no exceptions). For both alternatives we present the formula that expresses it and verify it on the model. Consider the test case *ProcessSaleCase*, which is represented by the following sequence of methods:

```
initializeCashDesk(0,0); startNewSale(); enterAllRemainingProducts();
finishSale(); handleCashPayment(); updateInventory();
```

Alternative (1). We say that the test passes if each *fair* run in the model, corresponding to the scenario given by the test case, comes to the end (is finite). This can be expressed by the formula

$$\psi_{PSC} \Rightarrow (\psi_{INF} \Rightarrow \neg \psi_{FAIR})$$

where

- ψ_{PSC} is satisfied iff the usage profile of the system corresponds to the sequence given by the *ProcessSaleCase*.
- $\psi_{INF} = \mathcal{G}\Big(\bigvee_{(n,\mathrm{act},m)\in SYST} \mathcal{P}(n, \mathrm{act}, m) \Big)$
  is satisfied on all runs on which all steps correspond to the steps of the system ($SYST$). Those runs model exactly the infinite runs of the system.
- $\psi_{FAIR} = \bigwedge_{(n,\mathrm{act},m)\in\mathcal{L}} \mathcal{G}\Big(\big(\mathcal{G}\, \mathcal{E}(n, \mathrm{act}, m)\big) \Rightarrow \big(\mathcal{F} \bigvee_{\mathrm{act}',m'} \mathcal{P}(n, \mathrm{act}', m')\big) \Big)$
  is satisfied on all fair runs. We say that a run is not fair iff in any of its states one of the components is able to send an action, but never does anything, because it is preempted by others.

Alternative (2). The test passes if it finishes in the sense of the alternative (1) and *no exception occurs* during the execution. This can be expressed by the formula

$$\psi_{PSC} \Rightarrow \Big(\neg\psi_{EXC} \wedge (\psi_{INF} \Rightarrow \neg\psi_{FAIR})\Big)$$

where

- $\psi_{EXC} = \mathcal{F}\Big(\bigvee_{(n,\mathrm{exc},m)\in E} \mathcal{P}(n,\mathrm{exc},m)\Big)$
  is satisfied on all runs where an exception occurs, E represents the labels for all exceptions.

The verification of the properties showed that the test alternative (2) fails, because exceptions may occur during the scenario, but (1) passes successfully.

1.6 Tools

To accomplish automated verification we use the DiVinE tool [2] (`http://anna.fi.muni.cz/divine/`) which implements distributed-memory LTL model-checking and state-space analysis. Our language (described in subsection 1.2.3, referred as the CoIn language) is however slightly different from the standard input language of DiVinE, therefore we actually use a modified version of DiVinE, with added support for CoIn models. This is not a part of the official release at this moment, but will be included in the coming release.

The input to the tool consists of two parts. The first is a text file with the model of the system. The file contains the specification of all basic components and the composition operations. The second part of the input is the property to be verified. It can be given either as a formula of CI-LTL (which is to be preprocessed into a property automaton and added to the input file) or a hand-written property automaton. The tool automatically verifies the model against the given property and reports its validity. If the property is not valid, the tool returns a run of the model that violates the property. The verification process itself is performed in parallel on a cluster of processors and hence the tool is capable of verifying even extremely large models.

1.7 Summary

The text discusses the capabilities of Component-interaction automata for creation of a detailed model of component-based system behaviour, which can be later on used for formal analysis and verification. Component-interaction automata are very general and can be used for modelling of various kinds of component-based systems.

In the CoCoME project, we have applied it to the modelling of a component-based system given as Java implementation. For this type of systems, we have defined the mapping of Java methods and events to the actions in the model, and we have identified three basic types of composition (the cube-like, star-like,

and handshake-like) that we have realized by the parameterizable composition operator the language provides.

During the modelling process, we have faced many modelling issues. These include exception handling, dynamic creation and destruction of instances, reflection of the internal state of a component, and simulation of the publish-subscriber communicational model. We have realized that thanks to the generality of the Component-interaction automata language, we are able to handle all of these in an elegant way.

For the analysis of the model, we have employed the parallel model checking tool DiVinE (see section 1.6), which is able to automatically verify system models, namely the communicational behaviour and interactions of individual components. The tool can be used both for checking the properties of the system and compliance of the model with the specification. The tool support for the analysis has shown to be very capable. However we currently miss tool support for the modelling phase. The models are written directly in the textual notation. In future, we aim to support the modelling with a user-friendly interface, or some automatization in form of transformation from other formalisms, which would allow other approaches to take advantage of our verification methods.

1. Černá, I., Vařeková, P., Zimmerova, B.: Component-interaction automata modelling language. Technical Report FIMU-RS-2006-08, Masaryk University, Faculty of Informatics, Brno, Czech Republic (2006)
2. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimecek, P.: Divine – a tool for distributed verification. In: Proceedings of the Computer Aided Verification conference (CAV'06). Volume 4144/2006 of LNCS., Seattle, WA, USA, Springer Berlin / Heidelberg (2006) 278–281
3. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction automata as a verification-oriented component-based system specification. In: Proceedings of the ESEC/FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), Lisbon, Portugal, Iowa State University, USA (2005) 31–38
4. The CoIn Team: The complete CoIn model of the Trading System (2007) http://anna.fi.muni.cz/coin/cocome/.
5. Vařeková, P., Moravec, P., Černá, I., Zimmerova, B.: Effective verification of systems with a dynamic number of components. In: Proceedings of the ESEC/FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS'07), Dubrovnik, Croatia, To appear (2007)
6. Černá, I., Vařeková, P., Zimmerova, B.: Component substitutability via equivalencies of component-interaction automata. In: Proceedings of the Workshop on Formal Aspects of Component Software (FACS'06), Prague, Czech Republic, Elsevier ENTCS (2006) 39–55
7. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, USA (2000)
8. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society Press (1977) 46–57